# Introduction to Diffusion Models in Python

## With Code Example

save for later

# What are Diffusion Models?

Diffusion models are a class of generative models that learn to gradually denoise data. They start with pure noise and iteratively refine it into high-quality samples. This process mimics the reverse of a diffusion process, where data is progressively corrupted with noise.

```python
import numpy as np
import matplotlib.pyplot as plt

def add_noise(image, noise_level):
    return image + np.random.normal(0, noise_level, image.shape)

def plot_noise_progression(image):
    fig, axes = plt.subplots(1, 5, figsize=(20, 4))
    noise_levels = [0, 0.2, 0.5, 0.8, 1.0]

    for i, noise in enumerate(noise_levels):
        noisy_image = add_noise(image, noise)
        axes[i].imshow(noisy_image, cmap='gray')
        axes[i].set_title(f'Noise: {noise}')
        axes[i].axis('off')

    plt.show()

# Example usage
image = np.random.rand(100, 100)
plot_noise_progression(image)
```

Swipe next ⟶

# The Diffusion Process

The diffusion process involves gradually adding noise to data over multiple timesteps. This process transforms complex data distributions into simple Gaussian noise. The goal of diffusion models is to learn and reverse this process.

```python
import torch
import torch.nn as nn

class DiffusionModel(nn.Module):
    def __init__(self, num_steps):
        super().__init__()
        self.num_steps = num_steps
        self.beta = torch.linspace(1e-4, 0.02, num_steps)
        self.alpha = 1 - self.beta
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)

    def forward_diffusion(self, x0, t):
        noise = torch.randn_like(x0)
        alpha_t = self.alpha_bar[t].reshape(-1, 1, 1, 1)
        return torch.sqrt(alpha_t) * x0 + torch.sqrt(1 - alpha_t) * noise, noise

# Example usage
model = DiffusionModel(num_steps=1000)
x0 = torch.randn(1, 3, 64, 64)  # Example image
t = torch.randint(0, 1000, (1,))
x_t, noise = model.forward_diffusion(x0, t)
```

Swipe next ⟶

save for later

# The Reverse Process

The reverse process is where the magic happens. It starts with pure noise and gradually denoises it to produce a sample. This is achieved by learning to predict and remove the noise at each step.

```
class Unet(nn.Module):
    def __init__(self):
        super().__init__()
        # Simplified U-Net architecture
        self.down1 = nn.Conv2d(3, 64, 3, padding=1)
        self.down2 = nn.Conv2d(64, 128, 3, padding=1)
        self.up1 = nn.ConvTranspose2d(128, 64, 3, padding=1)
        self.up2 = nn.ConvTranspose2d(64, 3, 3, padding=1)

    def forward(self, x, t):
        t = t.unsqueeze(-1).repeat(1, x.shape[1])
        x = torch.cat([x, t], dim=1)
        x1 = self.down1(x)
        x2 = self.down2(x1)
        x = self.up1(x2) + x1
        return self.up2(x)

model = Unet()
noise_pred = model(x_t, t.float())
```

Swipe next ⟶

# Training Objective

The training objective for diffusion models is to minimize the difference between the predicted noise and the actual noise added during the forward process. This is typically done using mean squared error (MSE) loss.

```
def diffusion_loss(model, x0, t):
    x_t, noise = model.forward_diffusion(x0, t)
    noise_pred = model(x_t, t.float())
    return nn.MSELoss()(noise_pred, noise)

# Training loop
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for epoch in range(100):
    for batch in dataloader:
        t = torch.randint(0, model.num_steps, (batch.shape[0],))
        loss = diffusion_loss(model, batch, t)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Swipe next ⟶

save for later

# Sampling from the Model

To generate new samples, we start with pure noise and iteratively apply the learned denoising process. This is typically done using a variant of Langevin dynamics or other sampling techniques.

```
@torch.no_grad()
def sample(model, n_samples, device):
    x = torch.randn(n_samples, 3, 64, 64).to(device)
    for t in reversed(range(model.num_steps)):
        t_batch = torch.full((n_samples,), t, device=device, dtype=torch.long)
        noise_pred = model(x, t_batch.float())
        alpha_t = model.alpha[t]
        alpha_bar_t = model.alpha_bar[t]
        beta_t = model.beta[t]

        if t > 0:
            noise = torch.randn_like(x)
        else:
            noise = torch.zeros_like(x)

        x = 1 / torch.sqrt(alpha_t) * (x - (1 - alpha_t) / torch.sqrt(1 -
alpha_bar_t) * noise_pred) + torch.sqrt(beta_t) * noise

    return x

samples = sample(model, n_samples=4, device='cuda')
```

Swipe next ⟶

# Conditioning Diffusion Models

Diffusion models can be conditioned on additional information to guide the generation process. This is useful for tasks like class-conditional generation or image-to-image translation.

```
class ConditionalUnet(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.unet = Unet()
        self.class_embed = nn.Embedding(num_classes, 64)

    def forward(self, x, t, class_label):
        t = t.unsqueeze(-1).repeat(1, x.shape[1])
        class_embed = self.class_embed(class_label).unsqueeze(-1).unsqueeze(-1)
        class_embed = class_embed.repeat(1, 1, x.shape[2], x.shape[3])
        x = torch.cat([x, t, class_embed], dim=1)
        return self.unet(x)

model = ConditionalUnet(num_classes=10)
x_t = torch.randn(4, 3, 64, 64)
t = torch.randint(0, 1000, (4,))
class_label = torch.randint(0, 10, (4,))
noise_pred = model(x_t, t.float(), class_label)
```

Swipe next ⟶

save for later

# Real-life Example: Image Inpainting

Diffusion models can be used for image inpainting, where the task is to fill in missing or corrupted parts of an image. Here's a simplified example of how this might work:

```
def inpaint(model, image, mask):
    # Assume image and mask are torch tensors
    noisy_image = image * mask + torch.randn_like(image) * (1 - mask)

    for t in reversed(range(model.num_steps)):
        t_batch = torch.full((1,), t, dtype=torch.long)
        noise_pred = model(noisy_image, t_batch.float())

        alpha_t = model.alpha[t]
        alpha_bar_t = model.alpha_bar[t]
        beta_t = model.beta[t]

        if t > 0:
            noise = torch.randn_like(noisy_image)
        else:
            noise = torch.zeros_like(noisy_image)

        noisy_image = 1 / torch.sqrt(alpha_t) * (noisy_image - (1 - alpha_t) /
torch.sqrt(1 - alpha_bar_t) * noise_pred) + torch.sqrt(beta_t) * noise
        noisy_image = image * mask + noisy_image * (1 - mask)

    return noisy_image

# Usage
image = load_image('damaged_image.png')
mask = create_mask(image)  # 1 for known pixels, 0 for unknown
inpainted_image = inpaint(model, image, mask)
```

Swipe next ⟶

# Real-life Example: Text-to-Image Generation

Diffusion models have shown impressive results in text-to-image generation. Here's a simplified example of how you might condition a diffusion model on text:

```
import torch
import clip

class TextConditionedDiffusion(nn.Module):
    def __init__(self):
        super().__init__()
        self.unet = Unet()
        self.clip_model, _ = clip.load("ViT-B/32", device="cuda")

    def forward(self, x, t, text):
        with torch.no_grad():
            text_features =
self.clip_model.encode_text(clip.tokenize(text).to('cuda'))

        t = t.unsqueeze(-1).repeat(1, x.shape[1])
        text_features = text_features.unsqueeze(-1).unsqueeze(-1).repeat(1, 1,
x.shape[2], x.shape[3])
        x = torch.cat([x, t, text_features], dim=1)
        return self.unet(x)

model = TextConditionedDiffusion()
x_t = torch.randn(1, 3, 256, 256).to('cuda')
t = torch.randint(0, 1000, (1,)).to('cuda')
text = ["A beautiful sunset over the ocean"]
noise_pred = model(x_t, t.float(), text)
```

save for later

# Understanding the Noise Schedule

The noise schedule determines how quickly noise is added during the forward process. It's crucial for the model's performance and typically follows a beta schedule.

```python
import numpy as np
import matplotlib.pyplot as plt

def linear_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02
    return np.linspace(beta_start, beta_end, timesteps)

def cosine_beta_schedule(timesteps, s=0.008):
    steps = timesteps + 1
    x = np.linspace(0, timesteps, steps)
    alphas_cumprod = np.cos(((x / timesteps) + s) / (1 + s) * np.pi * 0.5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return np.clip(betas, 0.0001, 0.9999)

timesteps = 1000
linear_betas = linear_beta_schedule(timesteps)
cosine_betas = cosine_beta_schedule(timesteps)

plt.plot(linear_betas, label='Linear')
plt.plot(cosine_betas, label='Cosine')
plt.title('Beta Schedules')
plt.xlabel('Timestep')
plt.ylabel('Beta')
plt.legend()
plt.show()
```

Swipe next ⟶

# Improving Sample Quality

Several techniques can improve the quality of samples from diffusion models. One popular method is Classifier-Free Guidance, which allows for better control over the generation process.

```
def classifier_free_guidance(model, x, t, class_labels, guidance_scale):
    # Unconditional
    noise_pred_uncond = model(x, t, torch.zeros_like(class_labels))

    # Conditional
    noise_pred_cond = model(x, t, class_labels)

    # Combine predictions
    return noise_pred_uncond + guidance_scale * (noise_pred_cond -
noise_pred_uncond)

# Usage in sampling loop
noise_pred = classifier_free_guidance(model, x, t, class_labels,
guidance_scale=3.0)
```

Swipe next ⟶

save for later

# Evaluating Diffusion Models

Evaluating generative models can be challenging. Common metrics include Fréchet Inception Distance (FID) and Inception Score (IS). Here's a simplified FID implementation:

```
from torchvision.models import inception_v3
import scipy

def calculate_activation_statistics(images, model):
    model.eval()
    activations = []

    with torch.no_grad():
        for batch in images:
            activations.append(model(batch))

    activations = torch.cat(activations, dim=0)
    mu = activations.mean(dim=0)
    sigma = torch.cov(activations.transpose(0, 1))

    return mu, sigma

def calculate_fid(real_images, generated_images):
    inception_model = inception_v3(pretrained=True, transform_input=False).cuda()
    inception_model = nn.Sequential(*list(inception_model.children())[:-1]).eval()

    mu1, sigma1 = calculate_activation_statistics(real_images, inception_model)
    mu2, sigma2 = calculate_activation_statistics(generated_images,
inception_model)

    diff = mu1 - mu2
    covmean, _ = scipy.linalg.sqrtm(sigma1.mm(sigma2), disp=False)

    if np.iscomplexobj(covmean):
        covmean = covmean.real

    fid = diff.dot(diff) + torch.trace(sigma1 + sigma2 - 2 * covmean)
    return fid.item()
```

Swipe next ⟶

# Challenges and Limitations

While powerful, diffusion models face challenges:

1. Slow sampling: The iterative nature of sampling can be time-consuming.
2. Training stability: Careful hyperparameter tuning is often necessary.
3. Mode collapse: The model may fail to capture the full diversity of the data distribution.

To address these, researchers are exploring techniques like:

```
def improved_sampling(model, n_samples, device, skip_steps=10):
    x = torch.randn(n_samples, 3, 64, 64).to(device)
    for t in reversed(range(0, model.num_steps, skip_steps)):
        t_batch = torch.full((n_samples,), t, device=device, dtype=torch.long)
        noise_pred = model(x, t_batch.float())

        # Apply multiple denoising steps at once
        for sub_t in reversed(range(max(0, t - skip_steps), t)):
            alpha_t = model.alpha[sub_t]
            alpha_bar_t = model.alpha_bar[sub_t]
            beta_t = model.beta[sub_t]

            x = 1 / torch.sqrt(alpha_t) * (x - (1 - alpha_t) / torch.sqrt(1 -
alpha_bar_t) * noise_pred)

            if sub_t > 0:
                x = x + torch.sqrt(beta_t) * torch.randn_like(x)

    return x
```

Swipe next ⟶

save for later

# Future Directions

Diffusion models are an active area of research with exciting potential. Future directions include improved architectures for faster sampling and better quality, application to new domains like 3D generation and video synthesis, and integration with other AI techniques for more powerful generative systems.

```
class Diffusion3D(nn.Module):
    def __init__(self, resolution=32):
        super().__init__()
        self.resolution = resolution
        self.conv3d = nn.Conv3d(1, 64, kernel_size=3, padding=1)
        self.upsample = nn.Upsample(scale_factor=2, mode='trilinear')
        self.final_conv = nn.Conv3d(64, 1, kernel_size=1)

    def forward(self, x, t):
        t = t.view(-1, 1, 1, 1, 1).repeat(1, 1, self.resolution, self.resolution,
self.resolution)
        x = torch.cat([x, t], dim=1)
        x = F.relu(self.conv3d(x))
        x = self.upsample(x)
        return self.final_conv(x)

# Usage
model = Diffusion3D()
x = torch.randn(1, 1, 32, 32, 32)
t = torch.randint(0, 1000, (1,))
output = model(x, t.float())
```

Swipe next ⟶

# Ethical Considerations

As diffusion models become more powerful, it's crucial to consider their ethical implications. These models can potentially generate highly realistic fake content, raising concerns about misinformation and privacy. Researchers and practitioners must work on developing robust detection methods and ethical guidelines for the use of these technologies.

```
def ethical_check(generated_content, sensitive_content_detector):
    risk_score = sensitive_content_detector(generated_content)
    if risk_score > THRESHOLD:
        return False, "Generated content may be sensitive or inappropriate."
    return True, "Content passed ethical check."

# Pseudocode for usage
generated_image = sample_from_diffusion_model(model, prompt)
is_safe, message = ethical_check(generated_image, sensitive_content_detector)
if is_safe:
    publish(generated_image)
else:
    log_and_discard(generated_image, message)
```

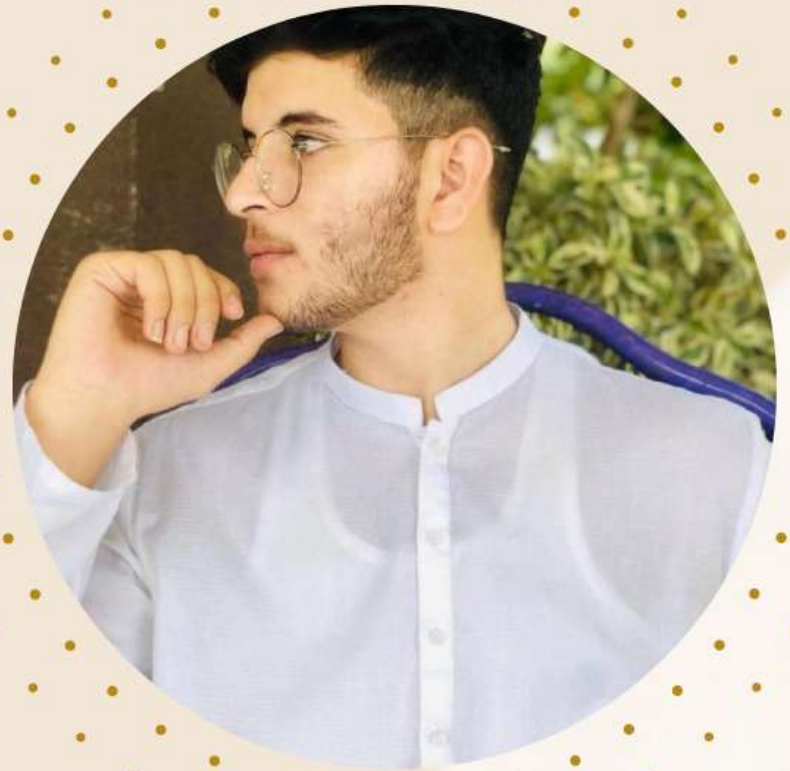
save for later

# Additional Resources

For those interested in diving deeper into diffusion models, here are some valuable resources:

1. "Denoising Diffusion Probabilistic Models" by Ho et al. (2020) ArXiv: https://arxiv.org/abs/2006.11239
2. "Diffusion Models Beat GANs on Image Synthesis" by Dhariwal and Nichol (2021) ArXiv: https://arxiv.org/abs/2105.05233
3. "High-Resolution Image Synthesis with Latent Diffusion Models" by Rombach et al. (2022) ArXiv: https://arxiv.org/abs/2112.10752

Data scientist
&ML Engineer



**Follow For More Data Science Content**